Diferencia entre int e Integer en Java

mario-alvial
15/10/2020

Conocer la diferencia entre **int** e **Integer** en JAVA puede ayudarnos mucho en nuestro día a día.

Estoy trabajando en la revisión de un código de un sistema para una tienda de autopartes online. En una de las revisiones encontramos la siguiente clase:

```
public class Cliente {
    private int edad;
    private String nombre;
}
```

En esta **clase**, mantenemos el nombre y la edad de cada cliente, pero el atributo de edad es opcional, el cliente no necesariamente debe proporcionarnos esta información. Entonces, ¿estás viendo algo extraño en esta clase?

Para facilitar la visualización, crearemos una **instancia** de esta clase e imprimiremos sus valores:

```
public static void main(String[] args) {
   Cliente cliente = new Cliente();
   System.out.println(cliente.getEdad());
```

```
    System.out.println(cliente.getNombre());
}
```

Si ejecutamos este código, tendremos el siguiente resultado:

```
0
null
```

Ahora que viene la parte extraña. ¿Por qué la edad es 0?. En ningun momento pasamos este valor. Y ademas, ¿edad "0"? Eso no existe en la práctica.

Comprendamos la razón de esto. Si volvemos a nuestra clase Cliente, podemos ver que el **atributo** edad es del tipo `int`. Este tipo, a su vez, se considera un **tipo primitivo**, que no es más que un tipo que no representa una clase.

Cuando declaramos que un atributo (miembro de la clase) es de algún tipo primitivo, ese atributo tiene un valor predeterminado. En nuestro caso, el valor predeterminado de int es 0, pero eso solo se aplica a los **atributos de una clase**.

Eso no es lo que queremos, ¿verdad?. Si el cliente registra la edad, queremos guardar la edad, de lo contrario, no queremos guardar ningún valor, es decir, queremos guardar el atributo de edad como vacío.

Clases Wrapper

Ahora, si pensamos bien, todos los tipos no primitivos son por defecto nulos, caso no no se les asigne ningún valor.

Considerando aquello, el lenguaje Java puso a nuestra disposición lo que llamamos **Wrapper** , que, básicamente, es una clase que representa un tipo primitivo. Por ejemplo, el Wraper de `int` es `Integer`. Tenga en cuenta que el Wrapper comienza con una letra mayúscula, sigue la misma nomenclatura que cualquier otra clase, porque también es una.

Pero, ¿qué obtengo usando este Wrapper?

Buena pregunta. Bueno, la mayor ventaja es poder asignar nulos. Pero con grandes poderes, vienen grandes responsabilidades, así que tenga cuidado de no asignar valores nulos sin critérios, ya que es muy probable que tu código tome **NullPointerException** .

También podemos hablar sobre los métodos que obtuvimos, ya que ahora estamos **escribiendo** nuestra variable con una clase, esa clase puede tener métodos que podemos usar para nuestro entero.

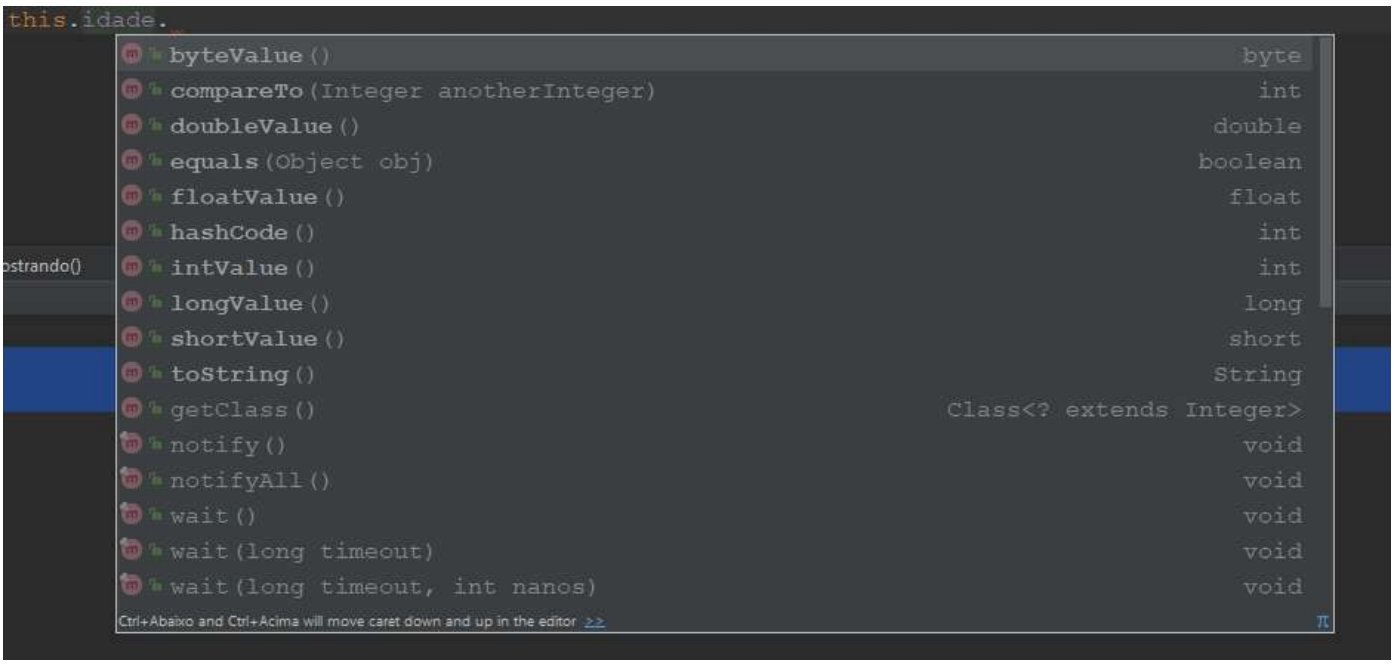
Para ejemplificar, intentemos ver los métodos disponibles para `edad`:

```
this.edad.
```

Pues ninguno, porque el tipo primitivo no es un objeto. No es una instancia de clase, por lo que no tiene métodos. Ahora cambiemos el tipo de edad a `Integer`:

```
private Integer edad;
```

Intentamos, de la misma manera, ver los métodos disponibles:



Además de los métodos, con Integer, también puedes agregar tu atributo a una colección, por ejemplo, puedes tener una Lista. Con tipos primitivos como int esto no es posible.

Parece que Integer es la solución a todos lo problemas

Sin embargo, es importante conocer algunas desventajas que los Wrappers tienen en relación a su tipo primitivo correspondiente.

Primero, los métodos mostrados arriba no se usan ampliamente. Además de estos métodos, tenemos los métodos estáticos, que son los más utilizados de las **clases wrappers**, pero tampoco son muy utilizados en la vida cotidiana, tal vez con excepción del

método `valueOf()`, que transforma el parámetro para el contenedor que llamó a este método.

Además, si es realmente necesario usar uno de los **métodos no estáticos**, la forma de transformar un `int` en un `Integer` es trivial.

Para lograr esta transformación, podemos usar exactamente este método `valueOf()`:

```
int tipoPrimitivo = 0;
Integer tipoPrimitivoTransformadoEnWrapper = Integer.valueOf(tipoPrimitivo);
```

O, dado que `Integer` es una clase no abstracta, podemos instanciarla:

```
int tipoPrimitivo = 0;
Integer tipoPrimitivoTransformadoEnWrapper = new Integer(tipoPrimitivo);
```

Concuerdo que es un poco aburrida la forma en como se realiza esta conversión. Es redundante tener que seguir dando **new** o llamando al método estático para obtener un número `Integer`...

Autoboxing y Unboxing

Al ver la regularidad de la conversión del tipo primitivo a Wrapper y viceversa, se desarrollaron dos funcionalidades para Java 5, **autoboxing** y **unboxing**.

Autoboxing es una característica que convierte automáticamente los tipos primitivos a su wrapper equivalente. Por ejemplo, supongamos que quiero transformar el atributo `edad` que tiene el tipo `int` en una nueva variable de tipo `Integer`. Con autoboxing podría hacer esto:

```
Integer edadWrapper = edad;
```

Y listo. ¿Genial, verdad? *"en backstages"* el compilador compila para el siguiente código:

```
Integer edadWrapper = Integer.valueOf(edad);
```

Y unboxing es el inverso del autoboxing, es decir, selecciona un wrapper y lo transforma en su tipo primitivo. Así:

```
int edadPrimitivo = edadWrapper;
```

"En backstages", lo que realiza es lo siguiente:

```
int edadPrimitivo = edadWrapper.intValue();
```

Ahora que entendemos mejor las diferencias entre `int` e `Integer`, podemos terminar de revisar la clase `Cliente`. Dado que edad es un atributo opcional y ciertamente no queremos que sea 0 por defecto, cambiemos su tipo de `int` a `Integer`. Obteniendo la siguiente clase:

```
public class Cliente {
    private Integer edad;
    private String nombre;
}
```

Operaciones matemáticas con Wrapper.

Faltó hablar de una cosa. ¿Y para operaciones matemáticas? ¿Qué es mejor usar? ¿ `int` o `Integer`?

Para esto crearemos un método para calcular el valor de las cuotas de compras realizadas en la tienda online. En este cálculo tenemos que seguir algunas reglas:

- Si la cantidad de prendas compradas es mayor que 3, se otorga un descuento del 10% sobre la compra total.
- La comisión del vendedor es igual al 5% del monto total de la compra después de los descuentos dados.
- El valor de las cuotas se adquiere dividiendo el total ya descontado por el número de cuotas.

Bueno, podemos crear un método que reciba el valor total, el número de prendas compradas, el número de cuotas y que devuelva un objeto de tipo `GerenciadorDeVentas` que recibe en su constructor el valor de la comisión del vendedor y el valor de cada paquete:

```
public GerenciadorDeVentas calculaVenta(double total, int numeroDePrendas, int
```

```
}
```

Como el foco no es la operación en sí, si resolviéramos este método, podríamos hacerlo de la siguiente manera:

```
public GerenciadorDeVentas calculaVenta(double total, int numeroDePrendas, int
   if(numeroDePrendas > 3){
      total *= 0.90;
   }
   double comision = total * 0.05;
   double valorCuotas = total / numeroDePrendas;
   return new GerenciadorDeVentas(comision, valorCuotas);
}
```

Ahora debes preguntarte por qué usé tipos primitivos en lugar de Wrappers, ¿verdad?. Bueno, si los usamos, correríamos el riesgo de recibir algún valor nulo y por lo tanto de recibir un `NullPointerException`. Entonces, usando tipos primitivos estamos evitando esto. Como máximo tendremos algunos de los parámetros con un valor de 0.

La segunda razón es la cuestión del rendimiento. Un Wrapper sigue siendo una clase y cuando creamos uno, de hecho, estamos creando instancias de un objeto, por lo que estamos reservando un espacio de memoria para él.

Pensando en ello, imagine la cantidad de objetos que tendríamos para resolver esta simple operación matemática, simplemente no vale la pena, porque todo lo que necesitamos es trabajar con valores y, en este aspecto, el tipo primitivo puede satisfacer esta necesidad.

Otro punto importante es que Java tiene una memoria caché para números bajos, por ejemplo:

```
public static void main(String[] args) {
   Integer i1 = 500;
   Integer i2 = 500;
   if(i1 == i2){
      System.out.println(true);
```

```
    }else{
       System.out.println(false);
    }
 }
```

Este código imprime `false` porque `Integer` es una clase, por lo que i1 e i2 son sus objetos y sus referencias se asignan en diferentes espacios de la memoria. Cuando usamos el operador == en los objetos, compara referencias y no valores. Es decir, como los objetos no apuntan a la misma referencia, tenemos `false`.

Ahora, mira este código:

```
 public static void main(String[] args) {
    Integer i1 = 127;
    Integer i2 = 127;
    if(i1 == i2){
       System.out.println(true);
    }else{
       System.out.println(false);
    }
 }
```

Esto, a su vez, imprime `true`.

Pero acabas de decir que comparar objetos usando el operador == resulta en `false`

Java, para ahorrar memoria, tiene un caché de algunos objetos, incluido `Integer`. Dependiendo de como se use, si los objetos Integer tienen un valor entre -128 y 127, Java puede hacer un *boxing* de los valores y reutilizarlos.

Por lo tanto, hacer comparaciones usando el operador == con Wrappers es arriesgado, porque además de comparar referencias en lugar de valores, podemos caer en este caso, donde se usa el caché y dar como resultado una comparación inesperada.

Dos soluciones a esto son: usar tipos primitivos para comparar usando "==" o usar el método equals (), que es más adecuado para comparar objetos.

Conclusión

Los wrappers, como `Integer`, son útiles cuando necesitamos usar nuestra variable en colecciones o queremos dejar algún atributo opcional, read, con un valor nulo. Los tipos primitivos son excelentes para cuando no queremos nulos y para operaciones matemáticas, ya que ocupan poco espacio en la memoria, mejorando el rendimiento de su aplicación.

Eso es todo. Espero que hayas logrado entender la diferencia entre `int` e`Integer` y cuándo vale la pena usar cada uno. ¿Ya has sufrido resultados inesperados al comparar un Wrapper?

En Alura tenemos toda una formación con varios cursos de **Java**, desde lo más básico con el lenguaje hasta conceptos más avanzados como clases, poliformismo y herencia. Pero no te asustes con esos términos, ellos son abordados con un proyecto práctico que junto con nuestra metodología y didáctica hacen que el contenido sea mucho más fácil de ser aprendido.

ARTÍCULOS DE TECNOLOGÍA > PROGRAMACIÓN

- Videos y actividades 100% en Español
- Certificado de participación
- Estudia las 24 horas, los 7 días de la semana
- Foro y comunidad exclusiva para resolver tus dudas
- Acceso a todo el contenido de la plataforma por 6 meses

¡QUIERO EMPEZAR A ESTUDIAR!

Paga en moneda local en los siguientes países

ANUAL

US$79,90

un solo pago de US$79,90

- 218 cursos
- Videos y actividades 100% en Español

Nuevos cursos
cada semana

NAVEGACIÓN

PLANES
INSTRUCTORES
BLOG
POLÍTICA DE PRIVACIDAD
TÉRMINOS DE USO
SOBRE NOSOTROS
PREGUNTAS FRECUENTES

¡CONTÁCTANOS!

¡QUIERO ENTRAR EN CONTACTO!

BLOG

PROGRAMACIÓN
FRONT END
DATA SCIENCE
INNOVACIÓN Y GESTIÓN
DEVOPS

AOVS Sistemas de Informática S.A
CNPJ 05.555.382/0001-33

SÍGUENOS EN NUESTRAS REDES SOCIALES

ALIADOS

Empresa participante do SCALE UP ENDEAVOR